Thinking about Interfaces in Go

2018-07-21 Mark McDonnell 21 mins read

This post is going to explain the importance of interfaces, and the concept of programming to abstractions (using the Go programming language), by way of a simple example.

While treading what might seem like familiar ground to some readers, this is a fundamental skill to understand because it enables you to design more flexible and maintable services.

Interfaces in Go

An ‘interface’ is a contract which describes ***behaviour*** (not ***data***), and in Go it looks something like the following:

```
type Foo interface {
    Bar(s string) (string, error)
}
```

If an object in your code implements a `Bar` function, with the exact same signature (e.g. accepts a string and returns either a string or an error), then that object is said to ***implement*** the `Foo` interface.

An example of this would be:

```
type thing struct{}

func (l *thing) Bar(s string) (string, error) {
  ...
}
```

Now you can define a function that will accept that object, as long as it fulfils the `Foo` interface, like so:

```
func doStuffWith(thing Foo)
```

This is different to other languages, where you have to ***explicitly*** assign an interface type to an object, like with Java:

```
class testClass implements Foo
```

Because of this flexibility in how interfaces are 'applied', it also means that an object could end up implementing ***multiple*** interfaces.

For example, imagine we have the following two interfaces:

```
type Foo interface {
  Bar(s string) (string, error)
}

type Beeper interface {
  Beep(s string) (string, error)
}
```

We can define an object that fulfils ***both*** interfaces simply by implementing the functions they define:

```
type thing struct{}

func (l *thing) Bar(s string) (string, error) {
  ...
}

func (l *thing) Beep(s string) (string, error) {
  ...
}
```

> *Note: this is a bit of silly example, and so you'll notice the method signature for each type is effectively the same. Be careful when designing your interfaces, because in this case we could possibly combine these two interfaces into a single (more generic) interface.*

Name Your Interface Arguments

Consider the following interface:

```
type Mover interface {
  Move(context.Context, string, string) error
}
```

Do you know what the second and third arguments refer to and how the function will use them?

Now consider this refactored version where the arguments have names associated with them:

```
type Mover interface {
  Move(context.Context, source string, destination string) error
}
```

Now ***that*** is better, because we can clearly see what the expectations are: the second argument is the 'source' and the third argument is the ‘destination’.

Keep Interfaces Small

You’ll find in the Go Proverbs, the following useful tip:

> *The bigger the interface, the weaker the abstraction.*

The reason for this is due to how interfaces are designed in Go and the fact that an object can potentially support multiple interfaces.

By making an interface too big, we reduce an object’s ability to support it. Consider the following example:

```go
type FooBeeper interface {
  Bar(s string) (string, error)
  Beep(s string) (string, error)
}

type thing struct{}

func (l *thing) Bar(s string) (string, error) {
  ...
}

func (l *thing) Beep(s string) (string, error) {
  ...
}

type differentThing struct{}

func (l *differentThing) Bar(s string) (string, error) {
  ...
}

type anotherThing struct{}

func (l *anotherThing) Beep(s string) (string, error) {
  ...
}
```

In the above example we’ve defined a `FooBeeper` interface that requires two methods: `Bar` and `Beep`. Now if we look at the various objects we’ve defined `thing`, `differentThing` and `anotherThing` we’ll find:

- `thing`: fulfils the `FooBeeper` interface
- `differentThing`: does ***not*** fulfil the `FooBeeper` interface
- `anotherThing`: does ***not*** fulfil the `FooBeeper` interface

Alternatively, if we were to break the `FooBeeper` interface up into separate smaller interfaces (like we demonstrated earlier), then in our above example, the `differentThing` and `anotherThing` would become more re-usable.

That's ultimately what this Go proverb is suggesting: smaller interfaces allow for greater code reuse.

Accept Interfaces, Return Concrete Types

If your function accepts a concrete type then you've limited the consumers ability to provide different implementations.

Consider a function only accepting the concrete type `*os.File` instead of the `io.Writer` interface. Now try swapping out the `os.File` implementation in a test environment, you'll have a hard time vs mocking this using a struct that has the relevant interface methods.

Try to return concrete types instead of interfaces, as interfaces have a tendendency to add an unnecessary layer of indirection for consumers of your package (although we'll discover a few valid scenarios where returning an interface is more appropriate).

Don't Return Concrete Types

This is to keep you on your toes 😉

I want to highlight an important 'design' decision, which is: if your code returns a pointer to some data, then it means once that pointer has been passed around a few different functions, we now have multiple entities that are able to mutate that data.

So be careful about whether you return a value (immutable) vs a pointer (mutable) as it could help reduce confusion with regards to how the data is modified by your program.

Returning an interface in these cases ***could*** be an appropriate solution.

By this I mean: although you might return a pointer to a data structure, by defining an interface around the behaviours attached to that data structure, it means a caller of your function won't be able to access the internal fields of the struct but it ***can*** call the methods defined in the returned interface!

Another example might be that your function needs to return a different type depending on a runtime condition (`*cough* generics *cough*`). If that's the case, then returning an interface could again be an appropriate workaround to the lack of generics in the Go 1.x language.

The following code example highlights the principle:

```go
type ItemInterface interface {
        GetItemValue() string
}

type Item struct {
        ID int
}

type URLItem struct {
        Item
        URL string
}

type TextItem struct {
        Item
        Text string
}

func (ui URLItem) GetItemValue(){
        return ui.URL
}

func (ti TextItem) GetItemValue(){
        return ti.Text
}

func FindItem(ID int) ItemInterface {
  // returns either a URLItem or a TextItem
}
```

The `FindItem` could be an internal library function that attempts to locate an item via multiple data sources. Depending on which data source the item was found, the type returned will change.

In this instance returning an interface allows the consumer to not have to worry about the change in underlying data types.

> *Note: it's possible the returned types could be consolidated into a single generic type struct, which means we can avoid returning an interface, but it depends on the exact scenario/use case.*

Use Existing Interfaces

It's important to not 'reinvent the wheel' and to utilise existing interfaces wherever possible (otherwise you'll suffer from a condition known as 'interface pollution').

The golang toolchain offers a tool called Go Guru which helps you to navigate Go code.

It's a command line tool, but it's designed to be utilised from within an editor (like Atom or Vim etc).

Here is a list of the sub commands available:

```
callees         show possible targets of selected function call
callers         show possible callers of selected function
callstack       show path from callgraph root to selected function
definition      show declaration of selected identifier
describe        describe selected syntax: definition, methods, etc
freevars        show free variables of selection
implements      show 'implements' relation for selected type or method
peers           show send/receive corresponding to selected channel op
pointsto        show variables the selected pointer may point to
referrers       show all refs to entity denoted by selected identifier
what            show basic information about the selected syntax node
whicherrs       show possible values of the selected error variable
```

This can be really useful for identifying (for example) whether a new interface you've defined is similar to an existing interface.

To demonstrate this, consider the following example...

```
// this is a duplicate of fmt.Stringer interface
type stringit interface {
        String() string
}

type testthing struct{}

func (t testthing) String() string {
        return "a test thing"
}
```

The `stringit` interface I've defined is actually a duplication of the existing standard library interface `fmt.Stringer`.

So using Guru via my Vim editor I can see (when I have my cursor over the `testthing` struct and I call Guru) that this concrete type implements not only `stringit` but a few other interfaces...

```
/main.go:33.6-33.14:                                                struct type t
/usr/local/Cellar/go/1.10.3/libexec/src/fmt/print.go:62.6-62.13:     implements fm
/main.go:29.6-29.13:                                                implements st
/usr/local/Cellar/go/1.10.3/libexec/src/runtime/error.go:66.6-66.13: implements ru
```

Now whether you continue to define a new interface is up to you. There are actually quite a few places in the Go standard library where interfaces are duplicated for (what I believe to be) semantic reasoning, but otherwise if you don't need to make an explicit/semantic distinction, then I'd opt to reuse an existing interface.

> *Note: for more details on how to use Guru, see this gist.*

Don't Force Interfaces

If your code doesn't require interfaces, then don't use them.

No point making the design of your code more complicated for no reason. Consider the following code which returns an interface

> *Note: the following example is modified from a much older post by William Kennedy.*

```
package main

import "fmt"

// Server defines a contract for tcp servers.
type Server interface {
        Start()
}

type server struct{}

// NewServer returns an interface value of type Server
func NewServer() Server {
        return &server{}
}

// Start allows the server to begin to accept requests.
func (s *server) Start() {
        fmt.Println("start called")
}

func main() {
        s := NewServer()
        fmt.Printf("%+v (%T)\n", s, s)
        s.Start()
}
```

The use of an interface here is a bit pointless. We should instead just return a pointer to an exported version of the server struct because the user is gaining no benefits from an interface being returned by `NewServer` (see Don't Return Concrete Types for a possible use case for returning interfaces, but the above example is not one of them).

Upgrading Interfaces

If you use have an interface that's used by lots of people, how do you add a new method to it without breaking their code? The moment you add a new method to the interface, their existing code that handles the concrete implementation will fail.

Unfortunately there isn't a completely clean solution to this problem. In essence the original interface needs to stay untouched and we need to define a ***new*** interface that contains the new behaviour. Then the consumer's of an interface will continue to reference the original interface while using a type assertion within their functions for the new interface.

Below is an example of this problem in action:

```go
package main

import "fmt"

type foo interface {
        bar() string
        baz() string // new method added, which breaks the code
}

func doThing(f foo) {
        fmt.Println("bar:", f.bar())
}

type point struct {
        X, Y int
}

func (p point) bar() string {
        return fmt.Sprintf("p=%d, y=%d", p.X, p.Y)
}

func main() {
        var pt point
        pt.X = 1
        pt.Y = 2

        doThing(pt)
}
```

In the above code we can see we have added a new method `baz` to our `foo` interface which means the concrete implementation `pt` is no longer satisfying the `foo` interface as it has no `baz` method.

> *Note: I appreciate the example is a bit silly because we could just update the code to support the new interface, but we have to imagine a world where your interface is provided as part of a public package that is consumed by lots of users.*

To solve this problem we need an intermediate interface. The following example demonstrates the process. The steps are...

1. define a new interface containing the new method
2. add the method to the concrete type implementation
3. document the new interface and ask your interface consumers to type assert for it

```go
package main

import "fmt"

type foo interface {
	bar() string
}

type newfoo interface {
	baz() string
}

// We want a `foo` interface type, but if that valid type can also do the new
// behaviour, then we'll execute that behaviour...

func doThing(f foo) {
	if nf, ok := f.(newfoo); ok {
		fmt.Println("baz:", nf.baz())
	}
	fmt.Println("bar:", f.bar())
}

// Original concrete implementation...

type point struct {
	X, Y int
}

func (p point) bar() string {
	return fmt.Sprintf("p=%d, y=%d", p.X, p.Y)
}

// New concrete implementation of `point` struct (has the new method)...

type newpoint struct {
	point
}

func (np newpoint) baz() string {
	return fmt.Sprintf("np !!! %d, ny !!! %d", np.X, np.Y)
}

func main() {
        var pt point
        pt.X = 1
        pt.Y = 2

        doThing(pt)

        var npt newpoint
        npt.X = 3
        npt.Y = 4

        doThing(npt)
}
```

> *Note: again, the example is a bit silly in that we're handling everything within a single file, whereas in reality the consumer won't have access to the original interface/implementation code like we do here (so just use your imagination ☺).*

The output of the above code is as follows:

```
bar: p=1, y=2
baz: np !!! 3, ny !!! 4
bar: p=3, y=4
```

So we can see we called `doThing` and passed a concrete type that satisfied the `foo` interface and so that function called the `bar` method it was expecting to exist. Next we called `doThing` again but passed a different concrete type that not only satisfied the `foo` interface, but the `newfoo` interface and within `doThing` we type assert that the object passed in is not only a `foo` but a `newfoo`.

What would this look like in practice then? Well, if the Go standard library wanted to add a new method to the existing (and very popular) `net/http` package `ResponseWriter` interface: they would create a new interface with just the new behaviour defined, then they would document its existence and in that documentation they would explain that if your HTTP handler required the new behaviour, then you should type assert for it.

Imagine if the go standard library just updated the `ResponseWriter` with the new method? Lots and lots of existing HTTP server code would break as the concrete implementation that was passed through would not support that implementation.

In fact this is exactly what the go standard library authors have done with the `Flusher` and `Hijacker` interfaces. The following code demonstrates the use of a type assertion to access the additional behaviour defined by those interfaces:

```go
func(w http.ResponseWriter, r *http.Request) {
        io.WriteString(w, "This will arrive before... ")

        if fl, ok := w.(http.Flusher); ok {
                fl.Flush()
                time.Sleep(1 * time.Second)
        }

        io.WriteString(w, "...this bit does.")
}
```

Standard Library Interfaces

Imagine we have a function `process`, whose responsibility is to make a HTTP request and do something with the response data:

```go
package main

import (
        "fmt"
        "io/ioutil"
        "net/http"
)

func process(n int) (string, error) {
        url := fmt.Sprintf("http://httpbin.org/links/%d/0", n)

        resp, err := http.Get(url)
        if err != nil {
                fmt.Printf("url get error: %s\n", err)
                return "", err
        }

        defer resp.Body.Close()

        body, err := ioutil.ReadAll(resp.Body)
        if err != nil {
                fmt.Printf("body read error: %s\n", err)
                return "", err
        }

        return string(body), nil
}
```

```
func main() {
	data, err := process(5)
	if err != nil {
		fmt.Printf("\ndata processing error: %s\n", err)
		return
	}
	fmt.Printf("Success: %v", data)
}
```

We can see our `process` function accepts an integer, which is interpolated into the URL that is requested. We then use the `http.Get` function from the net/http package to request the URL.

The function then stringify's the response body and returns it. This is sufficient for a basic example, but in the real world this function would likely do lots more processing to the response data.

It may not be immediately obvious but there are already many instances where interfaces are being utilised. Let's break down the code and see what interfaces there are.

The `http.Get` function returns a pointer to a `http.Response` struct, and from within that struct we extract the `Body` field and pass it to `ioutil.ReadAll`.

The `Body` field's 'type' is set to the `io.ReadCloser` interface. If we look at that interface we'll see it's made up of ***nested*** interface types:

```
type ReadCloser interface {
    Reader
    Closer
}
```

If we now look at the `io.Reader` and `io.Closer` interfaces, we'll find:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Closer interface {
    Close() error
}
```

This means that for the response body object to be valid, it must support the `Read` and `Close` functions defined by these interfaces (the returned object will likely include other functions, but it needs `Read` and `Close` at a minimum).

The next thing that happens in the code is that we pass `http.Response.Body` to an input/output function called `ioutil.ReadAll`.

If we look at the signature of `ioutil.ReadAll` we'll see that it accepts a type of `io.Reader`, which we've seen already, and so this is another indication of why smaller interfaces enable re-usability.

What the `io.Reader` interface means for our code is that the input we provide to `ioutil.ReadAll` must support a `Read` function, and (because `http.Response.Body` implements the `io.ReadCloser` interface) we know it does implement that required function.

So already we've seen quite a few built-in interfaces being utilised to support the standard library code we're using. More importantly, you'll find the use of these interfaces (`io.ReadCloser`, `io.Reader`, `io.Closer` and others) are used ***everywhere*** in the Go codebase (highlighting again how small interfaces enable greater code re-usability).

Tight Coupling

Now there's an issue with the above code, specifically the `process` function, and that is we've tightly coupled the `net/http` package to the function.

What this means is that the `process` function has to intrinsically ***know*** about HTTP and dealing with the various methods available to that package.

Also, if we want to test this function we're going to have a harder time because the `http.Get` call would need to be mocked somehow. We don't want our test suite to have to rely on a stable network connection or the fact that the endpoint being requested might be down for maintenance.

The solution to this problem is to invert the responsibility of the `process` function, also known as ‘dependency injection’. This is the basis of one of the S.O.L.I.D principles: ‘inversion of control’.

Dependency Injection

If we call a function, then it is our responsibility to provide it with all the things it needs in order to do its job.

In the case of our `process` function, it needs to be able to acquire data from somewhere (that could be a file, it could be a remote procedure call, it shouldn’t matter). The most important aspect to consider is ***how*** it acquires that data.

The ***how*** is not the responsibility of the `process` function, especially if we decide later on that we want to change the implementation from HTTP to GRPC or some other data source.

Meaning, we need to provide that functionality to the `process` function. Let’s see what this might look like in practice:

> *Note: this is just a first iteration, and is a poor design because although it shifts the problem slightly, there will still be tight coupling. I’ll come back to this code later and refactor away the coupling completely. The reason I’ve not done that upfront is because there are learnings to be had from trying to write tests for this code (which we’ll see in a minute).*

```
package main

import (
	"fmt"
	"io/ioutil"
	"net/http"
)

type dataSource interface {
	Get(url string) (*http.Response, error)
}

type httpbin struct{}
```

```go
func (l *httpbin) Get(url string) (*http.Response, error) {
	resp, err := http.Get(url)
	if err != nil {
		fmt.Printf("url get error: %s\n", err)
		return &http.Response{}, err
	}

	return resp, nil
}

func process(n int, ds dataSource) (string, error) {
	url := fmt.Sprintf("http:/httpbin.org/links/%d/0", n)

	resp, err := ds.Get(url)
	if err != nil {
		fmt.Printf("data source get error: %s\n", err)
		return "", err
	}

	defer resp.Body.Close()

	body, err := ioutil.ReadAll(resp.Body)
	if err != nil {
		fmt.Printf("body read error: %s\n", err)
		return "", err
	}

	return string(body), nil
}

func main() {
	data, err := process(5, &httpbin{})
	if err != nil {
		fmt.Printf("\ndata processing error: %s\n", err)
		return
	}
	fmt.Printf("\nSuccess: %v\n", data)
}
```

Refactoring Considerations

Let’s start by looking at the interface we’ve defined:

```go
type dataSource interface {
	Get(url string) (*http.Response, error)
}
```

We've not been overly explicit when naming this interface `dataSource`. Its name is quite generic on purpose so as not to imply an underlying implementation bias.

Unfortunately the defined `Get` method is still too tightly coupled to a specific implementation (i.e. it specifies `http.Response` as a return type).

Meaning, that although the refactored code is ***better***, it is far from perfect.

Next we define our own object for handling the implementation of the `Get` method, which internally is going to use `http.Get` to acquire the data:

```
type httpbin struct{}

func (l *httpbin) Get(url string) (*http.Response, error) {
	resp, err := http.Get(url)
	if err != nil {
		fmt.Printf("url get error: %s\n", err)
		return &http.Response{}, err
	}

	return resp, nil
}
```

By using this interface as the accepted type in the `process` function signature, we're going to be able to decouple the function from having to acquire the data, and thus allow testing to become much easier (as we'll see shortly), but the `process` function is still fundamentally coupled to HTTP as the underlying transport mechanism.

The reason this is a problem is because the `process` function still ***knows*** that the returned object is a `http.Response` because it has to reference the `Body` field of the response, which isn't defined on the object we've injected (meaning the function intrinsically ***knows*** of its existence).

How far you take your interface design is up to you. You don't necessarily have to solve all possible concerns at once (unless there really is a need to do so).

Meaning, this refactor ***could*** be considered 'good enough' for your use cases. Alternatively your values and standards may differ, and so you need to consider your options for how you might what to design this solution in such a way that it would allow the code to not be so reliant on HTTP as the transport mechanism.

> *Note: we'll revisit this code later and consider another refactor that will help clean up this first pass of code decoupling.*

But first, let's look at how we might want to test this initial code refactor (as testing this code allows us to learn some interesting things when it comes to needing to mock interfaces).

Testing

Below is a simple test suite that demonstrates how we're now able to construct our own object, with a stubbed response, and pass that to the `process` function:

```go
package main

import (
	"bytes"
	"io/ioutil"
	"net/http"
	"testing"
)

type fakeHTTPBin struct{}

func (l *fakeHTTPBin) Get(url string) (*http.Response, error) {
	body := "Hello World"

	resp := &http.Response{
		Body:          ioutil.NopCloser(bytes.NewBufferString(body)),
		ContentLength: int64(len(body)),
		StatusCode:    http.StatusOK,
		Request:       &http.Request{},
	}

	return resp, nil
}

func TestBasics(t *testing.T) {
	expect := "Hello World"
	actual, _ := process(5, &fakeHTTPBin{})

	if actual != expect {
```

```
                t.Errorf("expected %s, actual %s", expect, actual)
        }
}
```

Much like we do in the real implementation, we define a struct (in this case we've named it more explicitly) `fakeHTTPBin`.

The difference now, and what allows us to test our code is that we're manually creating a `http.Response` object with dummy data.

One part of this code that requires some extra explanation would be the value assigned to the response `Body` field:

```
ioutil.NopCloser(bytes.NewBufferString(body))
```

If we remember from earlier:

> *The* `Body` *field's 'type' is set to the* `io.ReadCloser` *interface.*

This means when mocking the `Body` value we need to return something that has both a `Read` and `Close` method. So we've used `ioutil.NopCloser` which, if we look at its signature, we see returns an `io.ReadCloser` interface:

```
func NopCloser(r io.Reader) io.ReadCloser
```

The `io.ReadCloser` interface is exactly what we need (as that interface indicates the returned concrete type will indeed implement the required `Read` and `Close` methods).

But to use it we need to provide the `NopCloser` function something that supports the `io.Reader` interface.

If we were to provide a simple string like `"Hello World"`, then this wouldn't implement the required interface. So we wrap the string in a call to `bytes.NewBufferString`.

The reason we do this is because the returned type is something that supports the `io.Reader` interface we need.

But that might not be immediately obvious when looking at the signature for `bytes.NewBufferString`:

```
func NewBufferString(s string) *Buffer
```

So yes it accepts a string, but we want an `io.Reader` as the return type, whereas this function returns a pointer to a `Buffer` type?

If we look at the implementation of `Buffer` though, we will see that it does actually implement the required `Read` function necessary to support the `io.Reader` interface.

Great! Our test can now call the `process` function and process the mocked dependency and the code/test works as intended.

> ***NOTE***: *Yes, we should probably use something more obvious and replace* `bytes.NewBufferString` *with something like* `bytes.NewReader`, `strings.NewReader`.

More flexible solutions?

OK, so we've already explained why this implementation might not be the best we could do. Let's now consider an alternative implementation:

```
package main

import (
        "fmt"
        "io/ioutil"
        "net/http"
)

type dataSource interface {
        Get(url string) ([]byte, error)
}

type httpbin struct{}

func (l *httpbin) Get(url string) ([]byte, error) {
        resp, err := http.Get(url)
        if err != nil {
                fmt.Printf("url get error: %s\n", err)
```

```
		return []byte{}, err
	}

	defer resp.Body.Close()

	body, err := ioutil.ReadAll(resp.Body)
	if err != nil {
		fmt.Printf("body read error: %s\n", err)
		return []byte{}, err
	}

	return body, nil
}

func process(n int, ds dataSource) (string, error) {
	url := fmt.Sprintf("http://httpbin.org/links/%d/0", n)

	resp, err := ds.Get(url)
	if err != nil {
		fmt.Printf("data source get error: %s\n", err)
		return "", err
	}

	return string(resp), nil
}

func main() {
	data, err := process(5, &httpbin{})
	if err != nil {
		fmt.Printf("\ndata processing error: %s\n", err)
		return
	}
	fmt.Printf("\nSuccess: %v\n", data)
}
```

All we've really done here is move more of the logic related to HTTP up into the `httpbin.Get` implementation of the `dataSource` interface. We've also changed the response type from `(*http.Response, error)` to `([]byte, error)` to account for these movements.

Now the `process` function has even ***less*** responsibility as far as acquiring data is concerned. This also means our test suite benefits by having a much simpler implementation:

```
package main

import "testing"

type fakeHTTPBin struct{}
```

```go
func (l *fakeHTTPBin) Get(url string) ([]byte, error) {
	return []byte("Hello World"), nil
}

func TestBasics(t *testing.T) {
	expect := "Hello World"
	actual, _ := process(5, &fakeHTTPBin{})

	if actual != expect {
		t.Errorf("expected %s, actual %s", expect, actual)
	}
}
```

Now our `fakeHTTPBin.Get` only has to return a byte array.

Conclusion

Is there more we can do to improve this code's design? Sure. But we'll leave a new refactor iteration to another post.

Hopefully this has given you a feeling for how interfaces are used in the Go standard library and how you might utilise custom interfaces yourself.

But before we wrap up… time (once again) for some self-promotion 🙊

code, guides

dependencies, go, interfaces

OpsBot: Operations Slackbot
Next →